10. What is the difference between Dispose and Finalize methods?

Brief summary: The Dispose method is used to free unmanaged resources. The Finalize method is the same thing as the destructor, so it's the method that is called on an object when it is being cleaned up by the Garbage Collector.

The Dispose method is used to free unmanaged resources. The Finalize method is the same thing as the destructor, so it's the method that is called on an object when it is being cleaned up by the Garbage Collector.

First, let's focus on the **Dispose** method. This method comes from the IDisposable interface and it is used to free up any unmanaged resources used by an object when its work is finished.

First of all, let's understand what managed and unmanaged resources are.

Managed resources, as their name suggests, are managed by the Common Language Runtime. Any objects we create with C# are managed resources. The Garbage Collector is aware of their existence, and once they are no longer needed it will free up the memory they occupy. That means we don't need to worry about managed resources cleanup as it is done automatically for us.

Unmanaged resources are beyond the realm of the CLR. The Garbage Collector doesn't know about them, so it will not perform any cleanup on them. Examples of unmanaged resources are database connections, file handlers, COM objects, opened network connections, etc. We as developers are responsible to perform the cleanup after we are done with those objects. If we don't, bad things may happen. For example, if we open a file to read it and we don't close it, the next attempt to open the same file will fail with an error saying that the file is currently in use.

If we have a class that uses some unmanaged resources, it should implement the IDisposable interface and provide an implementation of the Dispose method. The Dispose method should contain the code that cleans the unmanaged resource, for example, closes a file. Let's see a simple class that does so:

```
class FileReader : IDisposable
{
    private StreamReader? _streamReader;
    private readonly string _path;

    1 reference
    public FileReader(string path)
    {
        _path = path;
    }

    2 references
    public string ReadLine()
    {
        _streamReader ??= new StreamReader(_path);
        return _streamReader.ReadLine();
    }

    0 references
    public void Dispose()
    {
        _streamReader?.Dispose();
    }
}
```

This class is simply providing a way to read a file line-by-line. Please note that this implementation is simplified for example's sake so it doesn't contain any error handling. This class implements the IDisposable interface, and because of that, it is forced to provide the implementation of the Dispose method. In this method, we clean up any unmanaged resources. In this case, we simply call the Dispose method of the StreamReader. This is a very common practice when implementing the Dispose method because it rarely happens that we need to access unmanaged resources that do not have any C# class meant to use them provided. Calling Dispose method from a dependency of a class (or methods, if we have more IDisposable dependencies) in the Dispose method o this class is called a **cascade Dispose**.

All right. Let's use the FileReader class:

```
var fileReader = new FileReader("input.txt");
var line1 = fileReader.ReadLine();
var line2 = fileReader.ReadLine();
```

At the first glance, it may seem ok - line1 and line2 are set to values coming from the input.txt file. The problem here is that we do not actually call the Dispose method, and the StreamReader is never closed. Let's fix that. We could call the Dispose method manually:

```
var fileReader = new FileReader("input.txt");
var line1 = fileReader.ReadLine();
var line2 = fileReader.ReadLine();
fileReader.Dispose();
```

…but this is a bit awkward and easy to forget, not to mention that if the exception will be thrown in this code, the Dispose method may never be called. It's better to use the **using statement**:

```
using (var fileReader = new FileReader("input.txt"))
{
    var line1 = fileReader.ReadLine();
    var line2 = fileReader.ReadLine();
}
```

Starting with C# 8 we can use the following syntax without braces:

```
using var fileReader = new FileReader("input.txt");
var line1 = fileReader.ReadLine();
var line2 = fileReader.ReadLine();
```

Remember, the using statement is just syntactic sugar for this:

```
FileReader fileReader = null;
try
{
    fileReader = new FileReader("input.txt");
    {
        var line1 = fileReader.ReadLine();
        var line2 = fileReader.ReadLine();
    }
}
finally
{
    fileReader?.Dispose();
}
```

The “finally” block is used to ensure that the Dispose method will be called no matter if the exception will be thrown or not.

All right. One more thing before we move on to the Finalize method. Remember that **the Garbage Collector does not call the Dispose method**. We must call it ourselves, and the best way to do it is by using the **using statement**, which ensures that the Dispose method will be called.

Now, let’s move to the **Finalize** method. This method is called on an object when it is being cleaned up by the Garbage Collector. That means it can only be added to reference types, so a struct or a record struct can’t have a finalizer defined. Please notice that in C# the destructor, the finalizer, and the Finalize method are the same things. We can’t even define the Finalize method in a class - we must do so by defining a destructor. Let’s see this in practice.

```
class Person
{
    2 references
    string Name { get; }
    1 reference
    public Person(string name) => Name = name;
    0 references
    ~Person()
    {
        Console.WriteLine($"Person {Name} is being destructed");
    }
}
```

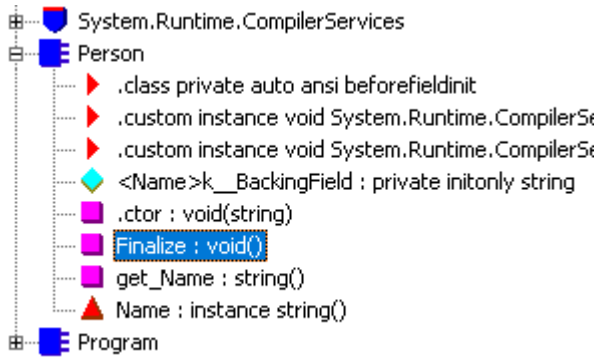
I defined a Person class that contains a destructor. When an object of this class will be cleaned up by the Garbage Collector this method will be executed. In my Main method, I run the following code:

```
SomeMethod();

GC.Collect();
Console.ReadKey();

void SomeMethod()
{
    var john = new Person("John");
}
```

john object lives in the scope of SomeMethod, and after this method finishes it is no longer needed. By running **GC.Collect** command I ask the Garbage Collector to do its work. And this is the result of the program:

```
Person John is being destructed
```

We said before that the Finalize method and the destructor are the same things. To prove it, let me show you how the Person class looks after being compiled into the Common Intermediate Language. I will use **ildasm** to read the dll.

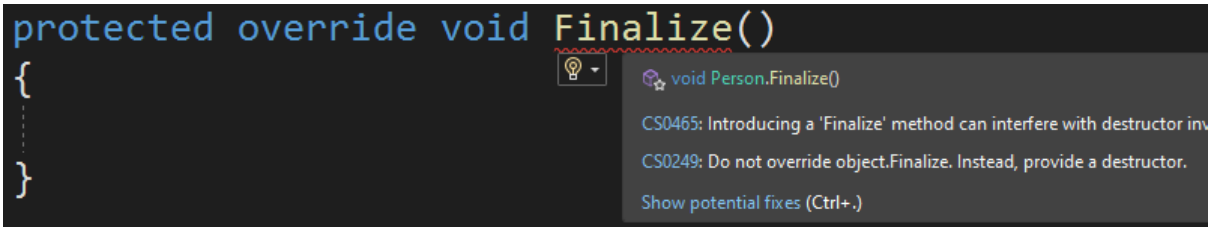


As you can see the Finalize method is added. And this is how it looks in CIL:

```
.method family hidebysig virtual instance void
        Finalize() cil managed
{
  .override [System.Runtime]System.Object::Finalize
  // Code size       40 (0x28)
  .maxstack  3
  IL_0000:  nop
  .try
  {
    IL_0001:  nop
    IL_0002:  ldstr      "Person "
    IL_0007:  ldarg.0
    IL_0008:  call       instance string Person::get_Name()
    IL_000d:  ldstr      " is being destructed"
```

As you can see it prints the same message as we defined in the destructor. In other words, the destructor is changed into the Finalize method during the compilation.

If I tried to add the Finalize method manually, I would get an error "Do not override object.Finalize. Instead, provide a destructor":



All right. We now know how to define destructors, so it's time to learn **when to use them**.

Well, the answer is "**almost never**" and I'm quoting Eric Lippert, one of the designers of C#. As we learned before, if a class is using some unmanaged resources that must be cleaned up, it should implement the IDisposable interface. Some people think that having a destructor that calls the Dispose method can be an assurance that those resources will be cleaned up if someone forgets to call the Dispose method manually or with the using statement. But this is solving an issue that shouldn't happen at all if developers know what they are doing, and in the process, we may cause much more problems. Let me quote Mr. Lippert again:

"If you make a destructor *be extremely careful* and *understand how the garbage collector works*. Destructors are *really weird*:

- They don't run on your thread; they run on their own thread. Don't cause deadlocks!
- An unhandled exception thrown from a destructor is bad news. It's on its own thread; who is going to catch it?
- A destructor may be called on an object *after* the constructor starts but *before* the constructor finishes. A properly written

destructor will not rely on invariants established in the constructor.
- A destructor can "resurrect" an object, making a dead object alive again. That's really weird. Don't do it.
- **A destructor might never run**; you can't rely on the object ever being scheduled for finalization. It *probably* will be, but that's not a guarantee.

Almost nothing that is normally true is true in a destructor. Be really, really careful.

Writing a correct destructor is very difficult."

Then, he also states that this was the only scenario when he needed to actually write destructors:

"When testing the part of the compiler that handles destructors. I've never needed to do so in production code."

The quote comes from this thread on Stack Overflow:
https://stackoverflow.com/questions/4898733/when-should-i-create-a-destructor/4899622

If you want to learn more about the tricky beasts that destructors are, make sure to read this article by Eric Lippert:
https://ericlippert.com/2015/05/18/when-everything-you-know-is-wrong-part-one/

So the bottom line here is: do not write destructors. If your objects must clean up some resources after they finish their work, make them implement the IDisposable interface.

Let's summarize. The Dispose method is used to free unmanaged resources. The Finalize method is the same thing as the destructor, so it's the method that is called on an object when it is being cleaned up by the Garbage Collector.

Bonus questions:

- "**What is the difference between a destructor, a finalizer, and the Finalize method?**"
  There is no difference, as they are the same thing. During the compilation, the destructor gets changed to the Finalize method which is commonly called a finalizer.

- "**Does the Garbage Collector call the Dispose method?**"
  No. The Garbage Collector is not aware of this method. We must call it ourselves, usually by using the using statement.

- "**When should we write our own destructors?**"
  The safest answer is "almost never". Destructors are very tricky and we don't even have a guarantee that they will run. Use IDisposable instead.

- "**What are managed and unmanaged resources?**"
  *The **managed resources** are managed by the Common Language Runtime. Any objects we create with C# are managed resources. The Garbage Collector is aware of their existence, and once they are no longer needed it will free up the memory they occupy. That means we don't need to worry about managed resources cleanup as it is done automatically for us. **Unmanaged resources** are beyond the realm of the CLR. The Garbage Collector doesn't know about them, so it will not perform any cleanup on them. Examples of unmanaged resources are database connections, file handlers, COM objects, opened network connections, etc. We as developers are responsible to perform the cleanup after we are done with those objects.*